Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Week 3 – Sequences and Iterators

Thomas Corbat / Felix Morgner

Rapperswil, 04.10.2022

HS2022

OST
Ostschweizer
Fachhochschule

Goals

- You can use `std::array` and `std::vector` in your code
- You can use an iterator to access container elements
- You can access and modify container contents using algorithms
- You can interact with streams through iterators

Topics

- **Recap Week 2**
- **Introduction to Basic Sequence Containers**
- **Iteration over Container Elements**
- **Introduction to Algorithms**
- **Iterators for I/O**

Recap Week 2

OST
Ostschweizer
Fachhochschule

- **What is the output?**

```
void main() -> int {
  std::cout << 42 << '\n'
            << std::oct << 42 << '\n';

  std::cout << std::setfill('\052') << std::setw(5) << 42 << "\n";

  std::cout << 42 << '\n’;
}
```

- **Which include is required for `std::setw`?**

- **Which incldues for IO are required and why?**

main.cpp

```cpp
#include "sayhello.h"
#include <?>

auto main() -> int{
  sayHello(std::cout);
}
```

sayhello.hpp

```cpp
#ifndef SAYHELLO_HPP_
#define SAYHELLO_HPP_

#include <?>

auto sayHello(std::ostream&) -> void;

#endif /* SAYHELLO_H_ */
```

sayhello.cpp

```cpp
#include "sayhello.h"
#include <?>

auto sayHello(std::ostream& os) -> void {
  os << "Hi there!\n";
}
```

`std::array` and `std::vector`

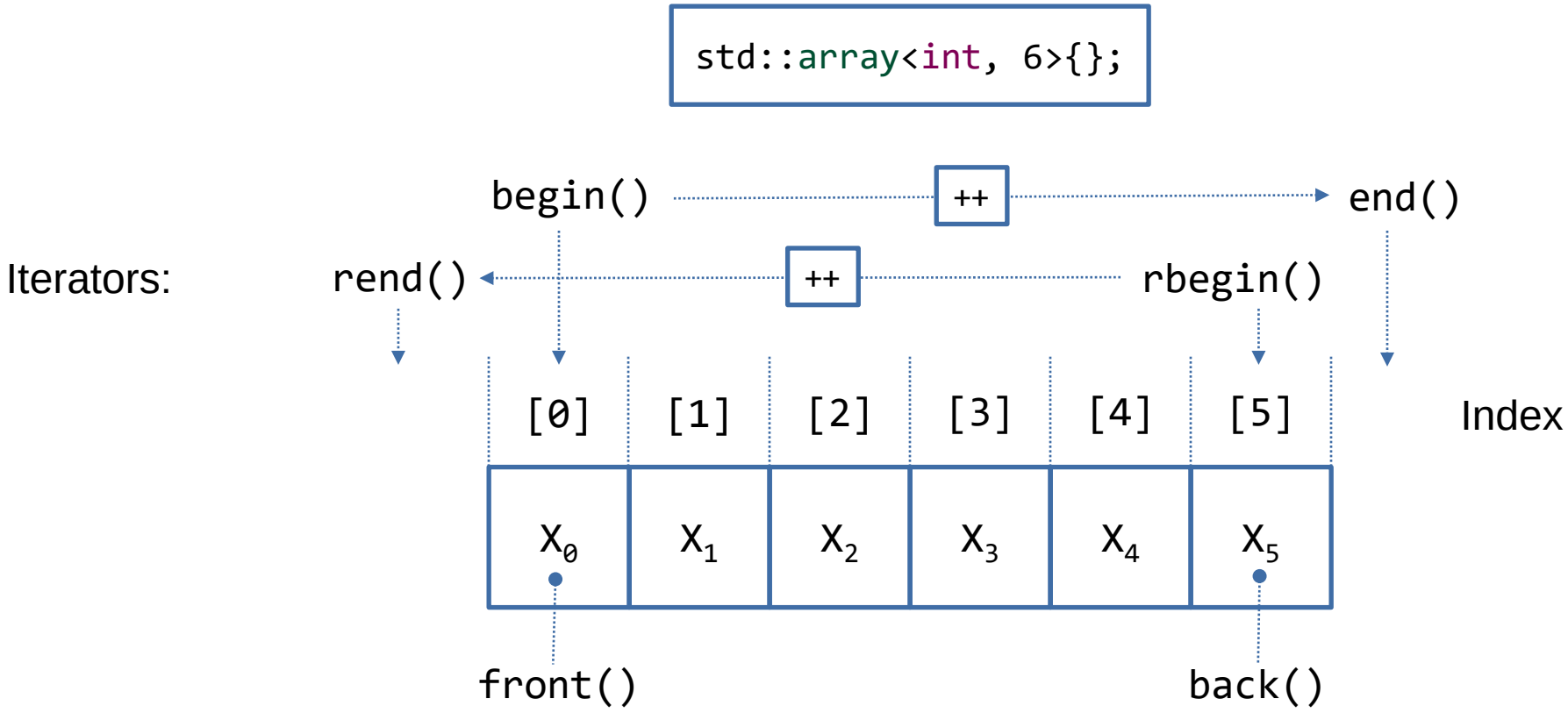
Goals:

- You can use **`std::array`** and **`std::vector`** in your code
- You know how to get iterators for standard containers

```
std::array<int, 5> name{1, 2, 3, 4, 5};
```

- **C++'s `std::array<T, N>` is a fixed-size Container**
  - `T` is a *template type parameter* (= placeholder for type)
  - `N` is a positive integer, *template non-type parameter* (= placeholder for a value)
  - Both can be deduced from the initializer
- **`std::array` can be initialized with a list of elements**
  - The size of an array must be known at compile-time and cannot be changed
  - Otherwise, it contains `N` default-constructed elements: `std::array<int, 5> emptyArray{};`
- **The size is bound to the array object and can be queried using `.size()`**
- **Avoid plain C-Array whenever possible: `int arr[]{1, 2, 3, 4, 5};`**

https://en.cppreference.com/w/cpp/container/array



- **Element access using subscript operator `[]` or `at()`**
  - `at()` throws an exception on invalid index access
  - `[]` has undefined behavior on invalid index access

```
std::vector<int> name{1, 2, 3, 4, 5};
```

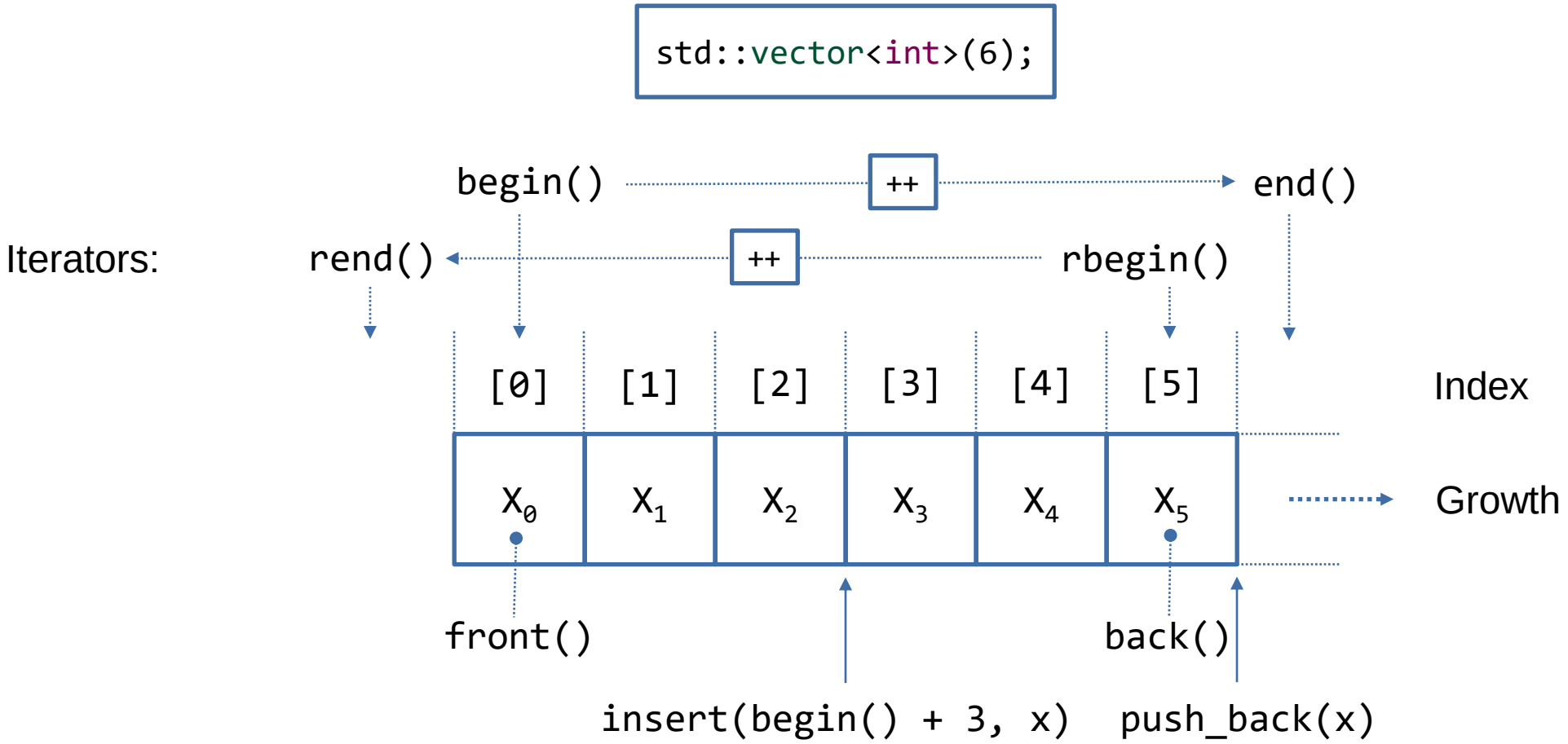
- **C++'s `std::vector<T>` is a Container = contains its elements of type T (no need to allocate them)**
  - `java.util.ArrayList<T>` is a collection = keeps references to T objects (must be “new”ed)
  - T is a *template type parameter* (= placeholder for type)
- **`std::vector` can be initialized with a list of elements**
  - The list can be empty: `std::vector<double> vd{};`
  - Other construction means might need parentheses (legacy)
- **When an initializer is given, the element type can be deduced!**

```
std::vector{1, 2, 3, 4, 5};
```

```
std::vector{};
```

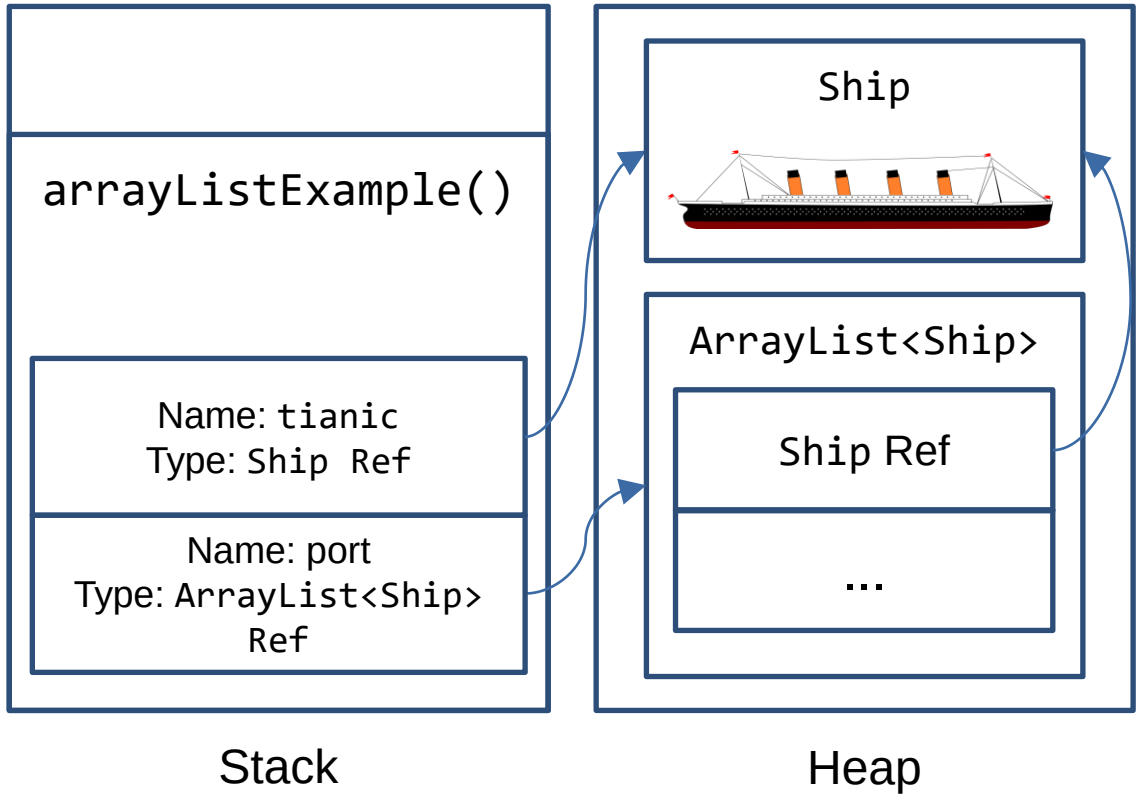
https://en.cppreference.com/w/cpp/container/vector

std::vector<T> Overview

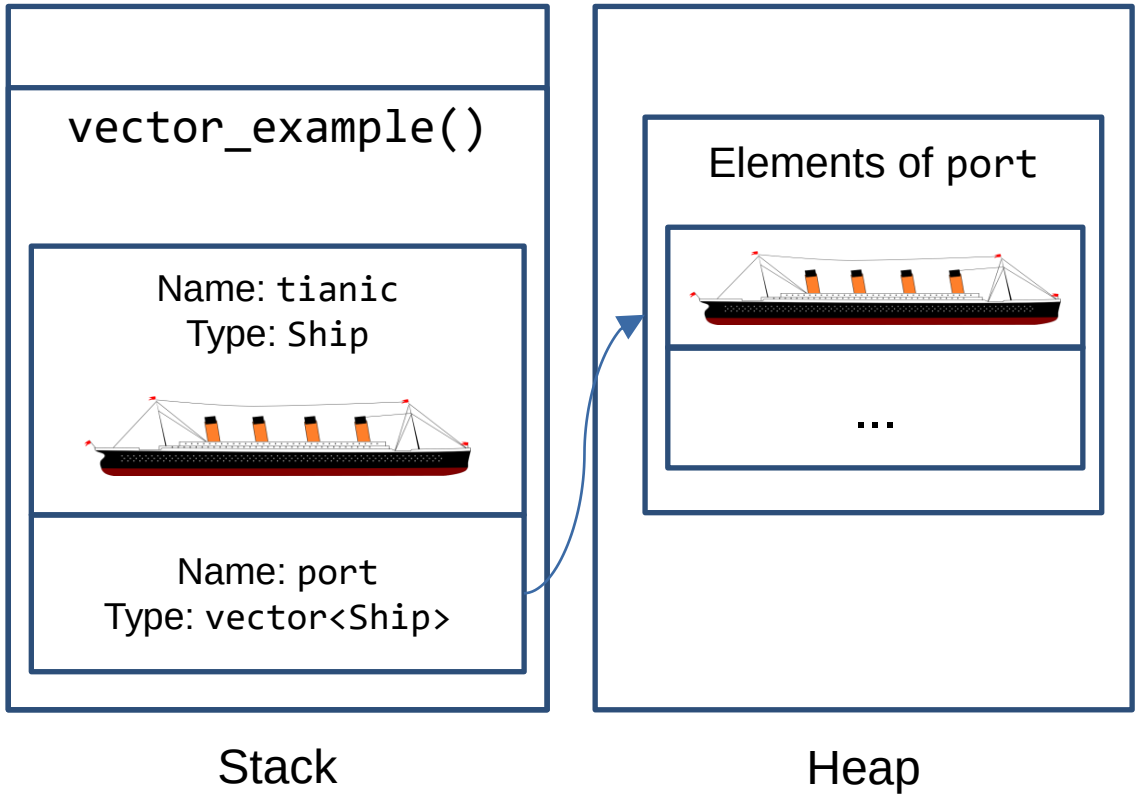


- **Parenthesis at definition allow providing initial size, when type of elements is a number**
  - `std::vector<std::string> words{6};` works

```java
public class SomeClassWeDontReallyNeed {
  public static void arrayListExample() {
    Ship titanic = new Ship("RMS Titanic");
    ArrayList<Ship> port = new ArrayList<>();
    port.add(titanic);
  }
}
```




```cpp
#include "Ship.hpp"
#include <vector>
auto vector_example() -> void {
  Ship titanic{"RMS Titanic"};
  std::vector<Ship> port{};
  port.push_back(titanic);
}
```



Iteration

Goals:

- You know how to get and use iterators from a standard container

```
for (size_t i = 0; i < v.size(); ++i) {
  std::cout << "v[" << i << "] = " << v[i] << '\n';
}
```

- **You can index a vector like an array**
  - CAUTION: No bounds check!
  - Accessing an element outside the valid range is Undefined Behavior

- **Index variable type is "unsigned"**
  - `std::size_t` or `std::vector<T>::size_type`
- **Accessing elements with `at()` checks bounds**
  - `std::out_of_range exception` is thrown when accessing an invalid index

```
for (size_t i = 0; i < v.size(); ++i) {
  std::cout << v.at(i) << '\n';
}
```

Bad Style Iteration – Problematic Example

- **Print all elements except the last**

```
void printButLast(std::vector<char> const & values) {
  for (size_t i = 0; i < values.size() - 1; ++i) {
    std::cout << "v[" << i << "] = " << values[i] << '\n';
  }
}

int main() {
  std::vector letters{'a', 'b', 'c', 'd'};
  printButLast(letters);

  std::vector<char> empty{};
  printButLast(empty);
}
```

- **Index-based iteration is used only if the actual index value is required!**

- **Advantage: No index error possible**
- **Works with all containers, even value lists `{1, 2, 3}`**

	const: • element cannot be changed	**non-const:** • element can be changed

reference:
• element in vector is accessed
	`for (auto const & cref : v) {`
	`  std::cout << cref << '\n';`
	`}`
		`for (auto & ref : v) {`
		`  ref *= 2;`
		`}`

copy:
• loop has own copy of the element
	`for (auto const ccopy : v) {`
	`  std::cout << ccopy << '\n';`
	`}`
		`for (auto copy : v) {`
		`  copy *= 2;`
		`  std::cout << copy << '\n';`
		`}`

```
for (auto it = std::begin(v); it != std::end(v); ++it) {
  std::cout << (*it)++ << ", ";
}
```

- **Start with `std::begin(v)`**
- **Compare against `std::end(v)`**
- **Access element with `*iterator`**
  - Changing the element in a non-const container is possible in this way
- **Guarantee to just have read-only access with `std::cbegin()` and `std::cend()`**

```
for (auto it = std::cbegin(v); it != std::cend(v); ++it) {
  std::cout << *it << ", ";
}
```

- **This kind of iteration is only useful if the position (the iterator) is required in the loop**

Using Iterators with Algorithms

Goals:

- You know some basic algorithms of the standard library
- You can apply the algorithms to an `std::vector` with its iterators

- **Each algorithm takes iterator arguments**
  - The range(s) of elements to apply an algorithm to is specified by iterators
- **The algorithm does what its name tells us**
- **Example: Counting values**
  - Algorithm `std::count` returns the number of occurrences of a value in range
  - Works with all ranges denoted by a pair of iterators

```
auto count_blanks(std::string s) -> size_t {
  size_t count{0};
  for (size_t i = 0; i < s.size(); ++i) {
    if (s[i] == ' ') {
      ++count;
    }
  }
  return count;
}
```

```
//The implementation is so simple it
//is not even necessary to create
//a separate function

auto count_blanks(std::string s) -> size_t {
  return std::count(s.cbegin(), s.cend(), ' ');
}
```

- **Summing up all values in a vector (with `std::accumulate`)**

```
#include <numeric>
```

  - Applies + operator to elements
  - Requires the initial value

```
std::vector<int> v{5, 4, 3, 2, 1};
std::cout << std::accumulate(std::cbegin(v), std::cend(v), 0)<< " = sum\n";
```

- **Number of elements in range (with `std::distance`)**

```
#include <iterator>
```

  - Containers provide a `size()` member function
  - Useful if you only have iterators

```
void printDistanceAndLength(std::string s) {
  std::cout << "distance: "<< std::distance(s.begin(), s.end()) <<'\n';
  std::cout << "in a string of length: "<< s.size()<<'\n';
}
```

`#include <algorithm>`


```
auto print(int x) -> void {
  std::cout << "print: "<< x << '\n';
}
auto printAll(std::vector<int> v) -> void {
  std::for_each(std::crbegin(v), std::crend(v), print);
}
```

- **Like `for` statement: Executes an action for each element in a range**
- **Last argument is a function ("first class value" in C++) that takes one parameter of the element type**
- **Using `std::cout` outside `main` is discouraged**
  - What can we do if we want to print to a given `std::ostream`?

```
auto print(int x, std::ostream & out) -> void {
  out << "print: "<< x << '\n';
}
auto printAll(std::vector<int> v, std::ostream & out) -> void {
  std::for_each(std::crbegin(v), std::crend(v), print(?, out));
}
```

```
auto printAll(std::vector<int> v, std::ostream & out) -> void {
  std::for_each(std::cbegin(v), std::cend(v), [&out](auto x) {
    out << "print: "<< x << '\n';
  });
}
```

Lambda structure:

```
[<capture>](<parameters>) -> <return-type> {
  <statements>
}
```

- A lambda expression creates a function object on the fly that can be passed to an algorithm
  - The created function is called from within the algorithm
  - Capture names variables taken from the surrounding scope, or define new ones (= copy, & -> reference, rename possible, type deduced)
- Parameters are like function parameters, if any, but you can use auto
- The return_type can be omitted if `void` or consistent return statements in the body (-> compiler knows)

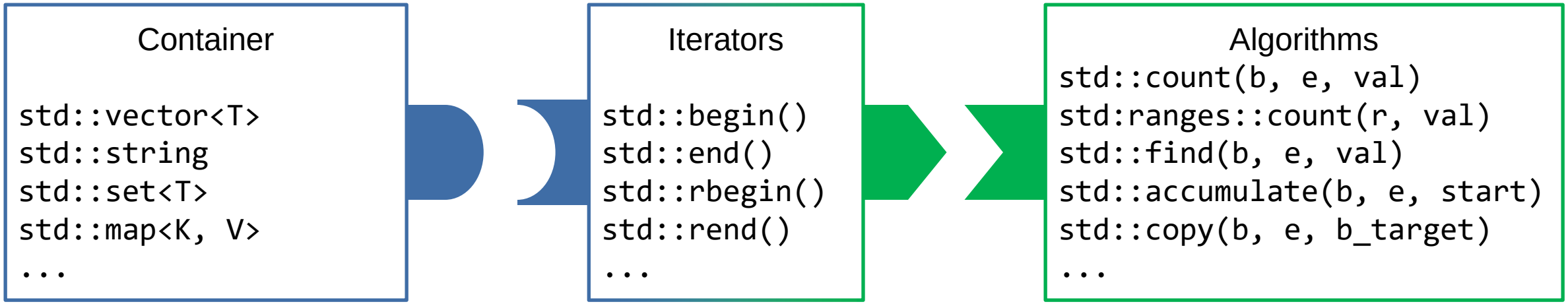
- **Often ranges supplied by iterators start at `begin()` and last until end()**
- **The Ranges specification introduce the namespace std::ranges, which contains many standard algorithms with a reduced parameter list**

Legacy algorithm

```
auto printAll(std::vector<int> v, std::ostream & out) -> void {
  std::for_each(std::cbegin(v), std::cend(v), [&out](auto x) {
    out << "print: "<< x << '\n';
  });
}
```

Range algorithm

```
auto printAll(std::vector<int> v, std::ostream & out) -> void {
  std::ranges::for_each(v, [&out](auto x) {
    out << "print: "<< x << '\n';
  });
}
```



- **Containers cannot be used with algorithms directly**
  - Iterators connect containers and algorithms

Appending Elements to an `std::vector<T>` (1/2)

- **Inserting elements into an `std::vector<T>`**
  - Append: `v.push_back(<value>);`
  - Insert anywhere: `v.insert(<iterator-position>, <value>);`

- **When using the `std::copy` algorithm the target has to be an iterator too**

```
std::copy(<input-begin-iterator>, <input-end-iterator>, <output-begin-iterator>);
std::ranges::copy(<input-range>, <output-begin-iterator>);
```

- **Can we do the following?**


```
std::vector<int> source{1, 2, 3}, target{};
std::copy(source.cbegin(), source.cend(), target.end());
std::ranges::copy(source, target.end());
```

Appending Elements to an `std::vector<T>` (2/2)

- **Inserting elements into an `std::vector<T>`**
  - Append: `v.push_back(<value>);`
  - Insert anywhere: `v.insert(<iterator-position>, <value>);`

- **When using the `std::copy` algorithm the target has to be an iterator too**

```
std::copy(<input-begin-iterator>, <input-end-iterator>, <output-begin-iterator>);
std::ranges::copy(<input-range>, <output-begin-iterator>);
```

- **We need an `std::back_inserter` or an `std::inserter`**

```
std::vector<int> source{1, 2, 3}, target{};
std::copy(source.cbegin(), source.cend(), std::back_inserter(target));
std::ranges::copy(source, std::back_inserter(target));
```

```
#include <algorithm>
```


- **Filling a vector with `std::fill` requires a vector with existing elements to be overwritten**

```
std::vector<int> v{};
v.resize(10);
std::fill(std::begin(v), std::end(v), 2);
```

```
std::ranges::fill(v, 2);
```

```
std::vector<int> v(10);
std::fill(std::begin(v), std::end(v), 2);
```

```
std::ranges::fill(v, 2);
```

Caution: Requires round parentheses in case of a vector with numeric elements, otherwise it would get 1 element whose value is 10

- **Or create a vector directly filled with 10 2s**
  - The element type is deduced to be `int` (from 2)

```
std::vector v(10, 2);
```

```
#include <iterator>
```


- **The algorithms `std::generate()` and `std::generate_n()` fill a range with computed values**
  - Either use `std::back_inserter` or a non-empty container

```
std::vector<double> powerOfTwos{};
double x{1.0};
std::generate_n(std::back_inserter(powerOfTwos),
                5,
                [&x] {return x *= 2.0;}
);
```

```
std::vector<double> powerOfTwos(5);
double x{1.0};
std::ranges::generate(powerOfTwos,
                      [&x] {return x *= 2.0;}
);
```

- **The `std::iota()` algorithm fills a range with subsequent values (1, 2, 3, ...)**

```
#include <numeric>
```

```
std::vector<int> v(100);
std::iota(std::begin(v), std::end(v), 1);
```

- **No ranges algorithms in <numeric> standard library header (until C++23)**

- **`std::(ranges::)find()` and `std::(ranges::)find_if()` return an iterator to the first element that matches the value or condition**
  - If no match exists the end of the range is returned

```
auto zero_it = std::ranges::find(v, 0);
if (zero_it == std::end(v)){
  std::cout << "no zero found \n";
}
```

- **Similarly `std::(ranges::)count()` and `std::(ranges::)count_if()` return the number of matching elements in a range**

```
std::cout << std::ranges::count(v, 42) << " times 42\n";
auto isEven = [](int x) { return !(x % 2); };
std::cout << std::ranges::count_if(v, isEven) << " even numbers\n";
```

Avoid Programming Your Own Loops (if possible)

- **Writing readable code is about expressing intentions**
  - For many intentions there is a matching iterator-based algorithm in the standard library
- **It is superior to use the corresponding algorithm (function call) instead of coding your own loop**
  - Correctness
  - Readability
  - Performance

```
bool find_with_loop(std::vector<int> const & values, int const v) {
  auto const end = std::end(values);
  for (auto it = std::begin(values); it != end; ++it) {
    if (*it == v) {
      return true;
    }
  }
  return false;
}
```

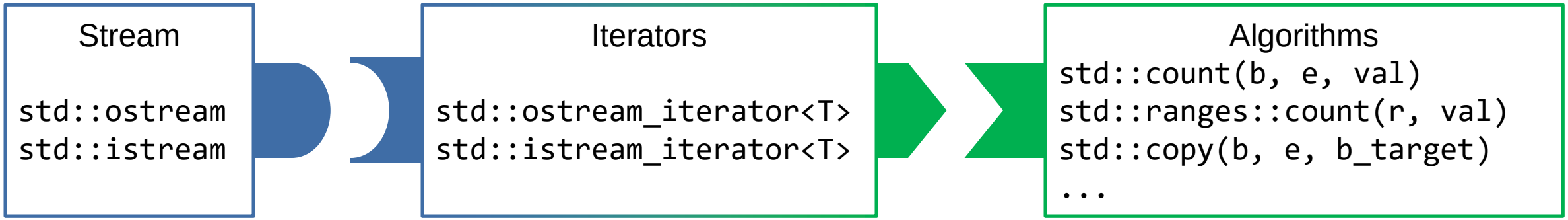
```
bool find_with_algorithm(std::vector<int> const & values, int const v) {
  auto const pos = std::ranges::find(values, v);
  return pos != std::end(values);
}
```

Iterators for I/O

Goals:

- You can create iterators for `std::istream`s and `std::ostream`s
- You can specify ranges on streams with stream iterators

Iterators Connect Streams and Algorithms



- **Streams (std::istream and std::ostream) cannot be used with algorithms directly**

```
std::ranges::copy(v, std::ostream_iterator<int>{std::cout, ", "});
```

- **`std::ostream_iterator<T>` outputs values of type `T` to the given `std::ostream`**
  - No `end()` marker needed for ouput, it ends when the input range ends

- **`std::istream_iterator<T>` reads values of type `T` from the given `std::istream`**
  - End iterator is the default constructed `std::istream_iterator<T>{}`
  - It ends when the stream is no longer `good()`

```
std::istream_iterator<std::string> in{std::cin};
std::istream_iterator<std::string> eof{};
std::ostream_iterator<std::string> out{std::cout, " "};
std::copy(in, eof, out);
```

- **`std::ranges::istream_view<T>` combines `in` and `eof`**

```
std::ranges::istream_view<std::string> in{std::cin};
std::ostream_iterator<std::string> out{std::cout, " "};
std::ranges::copy(in, out);
```

Shorter Type Names With Using Alias Declarations

- The (stream) iterators have a very unpleasant name length, even with auto-completion
- A type alias can help to abbreviate that

```
using <alias-name> = <type>;
```

- Useful if long type names occur more than once
- Example
  - Copy strings from standard input to standard ouput

```
using input = std::istream_iterator<std::string>;
input eof{};
input in{std::cin};
std::ostream_iterator<std::string> out{std::cout, " "};
std::copy(in, eof, out);
```

- **`std::istream_iterator` uses operator `>>` for input**
  - Disadvantage: It skips white space
- **For an exact copy, we also need the rest**
- **`std::istreambuf_iterator<char>` uses `std::istream::get()` to get every character**
  - This only works with `char`-like types

```
using input = std::istreambuf_iterator<char>;
input eof{};
input in{std::cin};
std::ostream_iterator<char> out{std::cout, " "};
std::copy(in, eof, out);
```

- **To fill a vector from a stream you can either use copy with `std::back_inserter(v)`**
  - It uses `v.push_back()` internally

```
using input = std::ranges::istream_view<int>;
std::vector<int> v{};
std::ranges::copy(input{std::cin}, std::back_inserter(v));
```

- **Or, construct the `std::vector<T>` directly from two iterators**

```
using input = std::istream_iterator<int>;
input eof{};
std::vector<int> const v{input{std::cin}, eof};
```

- **Prefer `std::array`/`std::vector` over plain C-Arrays**
- **Iterators specify ranges in C++**
- **Use algorithms over hand-written loops whenever possible**
- **Streams/containers require iterators to be processed by algorithms**
- **C++20 ranges simplify the use of algorithms**

- **Added range algorithm overloads**
- **Added ranges stream view**